

Exact gradient updates in time independent of output size for the spherical loss family

Pascal Vincent*, Alexandre de Brébisson, Xavier Bouthillier
 Département d'Informatique et de Recherche Opérationnelle
 Université de Montréal, Montréal, Québec, CANADA
 *and CIFAR

June 28, 2016

Abstract

An important class of problems involves training deep neural networks with sparse prediction *targets* of very high dimension D . These occur naturally in e.g. neural language models or the learning of word-embeddings, often posed as predicting the probability of next words among a vocabulary of size D (e.g. 200 000). Computing the equally large, but typically non-sparse D -dimensional output vector from a last hidden layer of reasonable dimension d (e.g. 500) incurs a prohibitive $O(Dd)$ computational cost *for each example*, as does updating the $D \times d$ output weight matrix and computing the gradient needed for backpropagation to previous layers. While efficient handling of large sparse network inputs is trivial, the case of large sparse *targets* is not, and has thus so far been sidestepped with approximate alternatives such as hierarchical softmax or sampling-based approximations during training. In this work we develop an original algorithmic approach which, for a family of loss functions that includes squared error and spherical softmax, can compute the *exact* loss, gradient update for the output weights, and gradient for backpropagation, all in $O(d^2)$ per example instead of $O(Dd)$, remarkably without ever computing the D -dimensional output. The proposed algorithm yields a speedup of $\frac{D}{4d}$, i.e. two orders of magnitude for typical sizes, for that critical part of the computations that often dominates the training time in this kind of network architecture.

1 Introduction

Many modern applications of neural networks have to deal with data represented, or representable, as very large sparse vectors. Such representations arise in natural language related tasks, where the dimension D of that vector is typically (a multiple of) the size of the vocabulary, but also in the sparse user-item matrices of collaborative-filtering applications. It is trivial to handle very large sparse inputs to a neural network in a computationally efficient manner: the forward propagation and update to the input weight matrix after backpropagation are correspondingly sparse. By contrast, training

with very large sparse prediction *targets* is problematic: even if the target is sparse, the computation of the equally large network output and the corresponding gradient update to the huge output weight matrix are *not sparse* and thus computationally prohibitive. This has been a practical problem ever since Bengio et al. [1] first proposed using a neural network for learning a language model, in which case the computed output vector represents the probability of the next word and is the size of the considered vocabulary, which is becoming increasingly large in modern applications [2]. Several approaches have been proposed to attempt to address this difficulty essentially by sidestepping it. They fall in two categories:

- *Sampling or selection based approximations* consider and compute only a tiny fraction of the output’s dimensions sampled at random or heuristically chosen. The reconstruction sampling of Dauphin et al. [3], the efficient use of biased importance sampling in Jean et al. [4], the use of Noise Contrastive Estimation [5] in Mnih and Kavukcuoglu [6] and Mikolov et al. [7] all fall under this category. As does the more recent use of approximate Maximum Inner Product Search based on Locality Sensitive Hashing techniques [8, 9] to select a good candidate subset.
- *Hierarchical softmax* [10, 7] imposes a heuristically defined hierarchical tree structure for the computation of the normalized probability of the target class.

Compared to the initial problem of considering all D output dimensions, both kinds of approaches are crude approximations. In the present work, we will instead investigate a way to actually perform the *exact* gradient update that corresponds to considering *all* D outputs, but do so implicitly, in a computationally efficient manner, without actually computing the D outputs. This approach works for a relatively restricted class of loss functions, that we call the *spherical family*, its simplest member being linear output with squared error (a natural choice for sparse real-valued regression targets). For simplicity and clarity we will begin with this squared error case, presenting the computational challenge that arises in the standard naive approach in Section 2 and deriving our algorithmic solution in Section 3. We will then extend our approach to the more general case of loss functions in the spherical family in Section 4. In Section 5 we will discuss numerical stability issues that may arise and detail our numerical stabilization strategy. Section 6 presents experimental validation focusing on timings obtained with our CPU and GPU implementations of our algorithm relative to the naive update algorithm.

2 The problem

2.1 Problem definition and setup

We are concerned with gradient-descent based training of a deep feed-forward neural network with target vectors of very high dimension D (e.g. $D = 200\,000$) but that are sparse, i.e. a comparatively small number, at most $K \ll D$, of the elements of the target vector are non-zero. Such a K -sparse vector will typically be stored and represented compactly as $2K$ numbers corresponding to pairs (*index*, *value*). A network to be trained with such targets will naturally have an equally large output layer of dimension D . We can also optionally allow the input to the network to be a similarly

high dimensional sparse vector of dimension D_{in} . Between the large sparse target, output, and (optionally large sparse) input, we suppose the network's intermediate hidden layers to be of smaller, more typically manageable, dimension $d \ll D$ (e.g. $d = 500$)¹.

Mathematical notation:

- Vectors are denoted using lower-case letters, e.g. h , and are considered column-vectors; corresponding row vectors are denoted with a transpose, e.g. h^T .
- Matrices are denoted using upper-case letters, e.g. W , with W^T the transpose of W .
- The j^{th} column of W is denoted W_j , and its i^{th} row $W_{i\bullet}$ (both viewed as a column vector).
- $U^{-T} = (U^{-1})^T$ denotes the transpose of the inverse of a square matrix.
- $\mathbf{1}_D$ denotes a D -dimensional column vector filled with ones.
- $\mathbf{1}_{i \in \mathcal{A}(y)}$ denotes an indicator function whose value will be 1 if $i \in \mathcal{A}(y)$ and 0 otherwise.
- $\text{onehot}_D(j) = \{\mathbf{1}_{i=j}\}_{i=1}^D$ is the D -dimensional column vector filled with zeros except at index j where its value is 1.
- \mathbf{I}_d is the $d \times d$ identity matrix.

Network architecture

We consider a standard feed forward neural network architecture as depicted in Figure 1. An input vector $x \in \mathbb{R}^{D_{in}}$ is linearly transformed into a linear activation $a^{(1)} = W^{(1)T}x + b^{(1)}$ through a $D_{in} \times d$ input weight matrix $W^{(1)}$ (and an optional bias vector $b^{(1)} \in \mathbb{R}^d$). This is typically followed by a non-linear transformation s to yield the representation of the first hidden layer $h^{(1)} = s(a^{(1)})$. This first hidden layer representation is then similarly transformed through a number of subsequent non-linear layers (that can be of any usual kind amenable to backpropagation) e.g. $h^{(k)} = s(a^{(k)})$ with $a^{(k)} = W^{(k)T}h^{(k-1)} + b^{(k)}$ until we obtain last hidden layer representation $h = h^{(m)}$. We then obtain the final D -dimensional network output as $o = Wh$ where W is a $D \times d$ output weight matrix, which will be our main focus in this work. Finally, the network's D -dimensional output o is compared to the D -dimensional target vector y associated with input x using squared error, yielding loss $L = \|o - y\|^2$.

Training procedure

This architecture is a typical (possibly deep) multi-layer feed forward neural network architecture with a *linear output layer* and *squared error loss*. Its parameters (weight matrices and bias vectors) will be trained by gradient descent, using gradient backpropagation Rumelhart et al. [11], LeCun [12, 13] to efficiently compute the gradients. The procedure is shown in Figure 1. Given an example from the training set as an *(input, target)* pair (x, y) , a pass of forward propagation proceeds as outlined above, computing the hidden representation of each hidden layer in turn based on the previous one, and finally the network's predicted output o and associated loss

¹Our approach does not impose any restriction on the architecture nor size of the hidden layers, as long as they are amenable to usual gradient backpropagation.

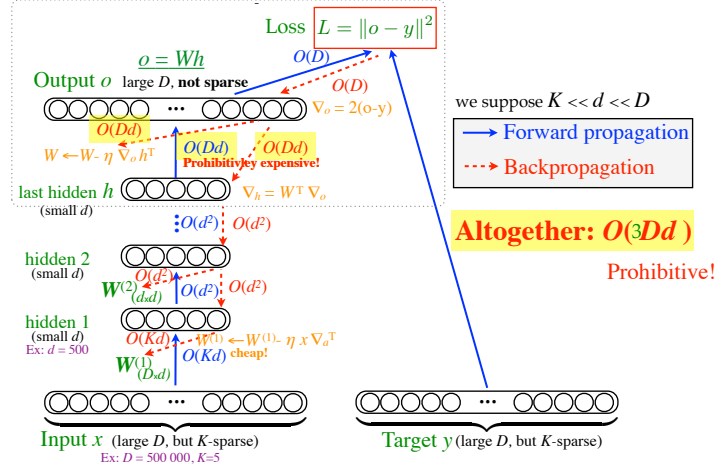


Figure 1: The computational problem posed by very large sparse targets. Dealing with sparse input efficiently is trivial, with both the forward and backward propagation phases easily achieved in $O(Kd)$. However this is not the case with large sparse targets. They incur a prohibitive computational cost of $O(Dd)$ at the output layer as forward propagation, gradient backpropagation and weight update each require accessing all $D \times d$ elements of the large output weight matrix.

L . A pass of gradient backpropagation then works in the opposite direction, starting from $\nabla_o = \frac{\partial L}{\partial o} = 2(o - y)$ and propagating back the gradients $\nabla_{h^{(k)}} = \frac{\partial L}{\partial h^{(k)}}$ and $\nabla_{a^{(k)}} = \frac{\partial L}{\partial a^{(k)}}$ upstream through the network. The corresponding gradient contributions on parameters (weights and biases), collected along the way, are straightforward once we have the associated $\nabla_{a^{(k)}}$. Specifically they are $\nabla_{b^{(k)}} = \nabla_{a^{(k)}}$ and $\nabla_{W^{(k)}} = h^{(k-1)}(\nabla_{a^{(k)}})^T$. Similarly for the input layer $\nabla_{W^{(1)}} = x(\nabla_{a^{(1)}})^T$, and for the output layer $\nabla_W = (o - y)h^T$. Parameters are then updated through a gradient descent step $W^{(k)} \leftarrow W^{(k)} - \eta \nabla_{W^{(k)}}$ and $b^{(k)} \leftarrow b^{(k)} - \eta \nabla_{b^{(k)}}$, where η is a positive learning-rate. Similarly for the output layer which will be our main focus here: $W \leftarrow W - \eta \nabla_W$.

2.2 The easy part: input layer forward propagation and weight update

It is easy and straightforward to efficiently compute the forward propagation, and the backpropagation and weight update part for the *input layer* when we have a very large D_{in} -dimensional but K -sparse input vector x with appropriate sparse representation. Specifically we suppose that x is represented as a pair of vectors u, v of length (at most) K , where u contains integer indexes and v the associated real values of the elements of x such that $x_i = 0$ if $i \notin u$, and $x_{u_k} = v_k$.

- **Forward propagation through the input layer:** The sparse representation of x as

the positions of K elements together with their value makes it cheap to compute $W^{(1)T}x$. Even though $W^{(1)}$ may be a huge full $D_{in} \times d$ matrix, only K of its rows (those corresponding to the non-zero entries of x) need to be visited and summed to compute $W^{(1)T}x$. Precisely, with our (u, v) sparse representation of x this operation can be written as $W^{(1)T}x = \sum_{k=1}^K v_k W_{:u_k}^{(1)}$ where each $W_{:u_k}^{(1)}$ is a d -dimensional vector, making this an $O(Kd)$ operation rather than $O(Dd)$.

- **Gradient and update through input layer:** Let us for now suppose that we were able to get gradients (through backpropagation) up to the first hidden layer activations $a^{(1)} \in \mathbb{R}^d$ in the form of gradient vector $\nabla_{a^{(1)}} = \frac{\partial L}{\partial a^{(1)}}$. The corresponding gradient-based update to input layer weights $W^{(1)}$ is simply $W^{(1)} \leftarrow W^{(1)} - \eta x (\nabla_{a^{(1)}})^T$. This is a rank-one update to $W^{(1)}$. Here again, we see that only the K rows of $W^{(1)}$ associated to the (at most) K non-zero entries of x need to be modified. Precisely this operation can be written as: $W_{:u_k}^{(1)} \leftarrow W_{:u_k}^{(1)} - \eta v_k \nabla_{a^{(1)}} \quad \forall k \in \{1, \dots, K\}$ making this again a $O(Kd)$ operation rather than $O(Dd)$.

2.3 The hard part: output layer propagation and weight update

Given some network input x we suppose we can compute without difficulty through forward propagation the associated last hidden layer representation $h \in \mathbb{R}^d$. From then on:

- Computing the final output $o = Wh$ incurs a prohibitive computational cost of $O(Dd)$ since W is a full $D \times d$ matrix. Note that there is a-priori no reason for representation h to be sparse (e.g. with a sigmoid non-linearity) but even if it was, this would not fundamentally change the problem since it is D that is extremely large, and we supposed d reasonably sized already. Computing the residual $(o - y)$ and associated squared error loss $\|o - y\|^2$ incurs an additional $O(D)$ cost.
- The gradient on h that we need to backpropagate to lower layers is $\nabla_h = \frac{\partial L}{\partial h} = 2W^T(o - y)$ which is another $O(Dd)$ matrix-vector product.
- Finally, when performing the corresponding output weight update $W \leftarrow W - \eta(o - y)h^T$ we see that it is a rank-one update that updates all $D \times d$ elements of W , which again incurs a prohibitive $O(Dd)$ computational cost.

For very large D , all these three $O(Dd)$ operations are prohibitive, and the fact that y is sparse, seen from this perspective, doesn't help, since neither o nor $o - y$ will be sparse.

3 A computationally efficient algorithm for performing the exact online gradient update

Previously proposed workarounds are approximate or use stochastic sampling. We propose a different approach that results in the *exact same*, yet efficient gradient update, remarkably without ever having to compute large output o .

3.1 Computing the squared error loss L and the gradient with respect to h efficiently

Suppose that, we have, for a network input example x , computed the last hidden representation $h \in \mathbb{R}^d$ through forward propagation. The network's D dimensional output $o = Wh$ is then in principle compared to the high dimensional target $y \in \mathbb{R}^D$. The corresponding squared error loss is $L = \|Wh - y\|^2$. As we saw in Section 2.3, computing it in the direct naive way would have a prohibitive computational complexity of $O(Dd + D) = O(Dd)$ because computing output Wh with a full $D \times d$ matrix W and a typically non-sparse h is $O(Dd)$. Similarly, to backpropagate the gradient through the network, we need to compute the gradient of loss L with respect to last hidden layer representation h . This is $\nabla_h = \frac{\partial L}{\partial h} = \frac{\partial \|Wh - y\|^2}{\partial h} = 2W^T(Wh - y)$. So again, if we were to compute it directly in this manner, the computational complexity would be a prohibitive $O(Dd)$. **Provided we have maintained an up-to-date matrix $Q = W^T W$** , which is of reasonable size $d \times d$ and can be cheaply maintained as we will see in Section 3.4, we can rewrite these two operations so as to perform them in $O(d^2)$:

Loss computation:

$$\begin{aligned}
 L &= \|\overbrace{Wh}^{O(Dd)} - y\|^2 \\
 &= (Wh - y)^T (Wh - y) \\
 &= h^T W^T W h - y^T W h - h^T W^T y + y^T y \\
 &= h^T Q h - 2h^T (W^T y) + y^T y \\
 &= h^T (\underbrace{Qh}_{O(d^2)} - 2\underbrace{W^T y}_{O(Kd)} + \underbrace{y^T y}_{O(K)})
 \end{aligned} \tag{1}$$

Gradient on h :

$$\begin{aligned}
 \nabla_h = \frac{\partial L}{\partial h} &= \frac{\partial \|Wh - y\|^2}{\partial h} \\
 &= 2W^T(Wh - y) \\
 &= 2(W^T W h - W^T y) \\
 &= 2(\underbrace{Qh}_{O(d^2)} - \underbrace{W^T y}_{O(Kd)})
 \end{aligned} \tag{2}$$

The terms in $O(Kd)$ and $O(K)$ are due to leveraging the K -sparse representation of target vector y . With $K \ll D$ and $d \ll D$, we get altogether a computational cost of $O(d^2)$ which can be several orders of magnitude cheaper than the prohibitive $O(Dd)$ of the direct approach.

3.2 Efficient gradient update of W

The gradient of the squared error loss with respect to output layer weight matrix W is $\frac{\partial L}{\partial W} = \frac{\partial \|Wh - y\|^2}{\partial W} = 2(Wh - y)h^T$. And the corresponding gradient descent update to W would be $W_{new} \leftarrow W - 2\eta(Wh - y)h^T$, where η is a positive learning rate. Again, computed in this manner, this induces a prohibitive $O(Dd)$ computational complexity, both to compute output and residual $Wh - y$, and then to update all the Dd elements of W (since generally neither $Wh - y$ nor h will be sparse). All $D \times d$ elements of W must be accessed during this update. On the surface this seems hopeless. But we will now see how we can achieve the *exact* same update on W in $O(d^2)$. The trick is to represent W *implicitly* as the factorization² $\underbrace{W}_{D \times d} = \underbrace{V}_{D \times d} \underbrace{U}_{d \times d}$ and update U and V instead:

$$\text{a) } U_{new} = U - 2\eta(Uh)h^T \quad (3)$$

$$\text{b) } V_{new} = V + 2\eta y(U_{new}^{-T}h)^T \quad (4)$$

This results in *implicitly* updating W as we did *explicitly* in the naive approach as we now prove:

$$\begin{aligned} V_{new}U_{new} &= (V + 2\eta y(U_{new}^{-T}h)^T)U_{new} \\ &= VU_{new} + 2\eta y(U_{new}^{-T}h)^T U_{new} \\ &= VU_{new} + 2\eta y h^T U_{new}^{-1} U_{new} \\ &= V(U - 2\eta(Uh)h^T) + 2\eta y h^T (U_{new}^{-1} U_{new}) \\ &= VU - 2\eta VUhh^T + 2\eta y h^T \\ &= VU - 2\eta(VUh - y)h^T \\ &= W - 2\eta(Wh - y)h^T \\ &= W_{new} \end{aligned}$$

We see that the update of U in Eq. 3 is a simple $O(d^2)$ operation. Following this simple rank-one update to U , we can use the Sherman-Morrison formula to derive the corresponding rank-one update to U^{-T} which will also be $O(d^2)$:

$$U_{new}^{-T} = U^{-T} + \frac{2\eta}{1 - 2\eta \|h\|^2} (U^{-T}h)h^T \quad (5)$$

It is then easy to compute the $U_{new}^{-T}h$, an $O(d^2)$ operation needed in Eq. 4. The ensuing rank-one update of V in Eq 4, thanks to the K -sparsity of y is only $O(Kd)$: only the K rows V associated to non-zero elements in y are accessed and updated, sited of *all* D rows of W we had to modify in the naive update!

²Note that we never *factorize* a pre-existing arbitrary W , which would be prohibitive as W is huge. We will no longer store a W nor work on it explicitly, but only matrices V and U which implicitly represent W .

3.3 Adapting the computation of L and ∇_h to the factored representation of W

With the factored representation of W as VU , we only have W implicitly, so the $W^T y$ terms that entered in the computation of L and ∇_h in the previous section (Eq. 1 on page 6 and 2 on page 6) need to be adapted slightly as $\hat{y} = W^T y = U^T (V^T y)$, which becomes $O(d^2 + Kd)$ rather than $O(Kd)$ in computational complexity. But this doesn't change the overall $O(d^2)$ complexity of these computations.

The adapted update computation of L and ∇_h can thus be expressed simply as:

$$\nabla_h = 2 \underbrace{\left(\underbrace{Qh}_{\hat{h}} - \underbrace{U^T(V^T y)}_{\hat{y}} \right)}_{\hat{z}} \quad (6)$$

and

$$L = h^T \underbrace{\left(\underbrace{Qh}_{\hat{h}} - 2 \underbrace{U^T(V^T y)}_{\hat{y}} \right)}_{\hat{z}} + y^T y \quad (7)$$

3.4 Bookkeeping: keeping an up-to-date Q and U^{-T}

We have already seen, in Eq. 5, how we can cheaply maintain an up-to-date U^{-T} following our update of U . Similarly, following our updates to U and V , we need to keep an up-to-date $Q = W^T W$ which is needed to efficiently compute the loss L (Eq. 1) and gradient ∇_h (Eq. 2). We have shown that updates to U and V in equations 3 and 4 are equivalent to implicitly updating W as $W_{new} \leftarrow W - 2\eta(W h - y)h^T$, and this translates into the following update to $Q = W^T W$:

$$Q_{new} = Q - \eta(h \nabla_h^T + \nabla_h h^T) + (4\eta^2 L) h h^T \quad (8)$$

One can see that this last bookkeeping operation also has a $O(d^2)$ computational complexity.

Proof that this update to Q corresponds to the update $W_{new} \leftarrow 2(W h - y)h^T$

$$\begin{aligned} W_{new}^T W_{new} &= (W - 2\eta(W h - y)h^T)^T (W - 2\eta(W h - y)h^T) \\ W_{new}^T W_{new} &= W^T W - 2\eta h(W h - y)^T W - 2\eta W^T (W h - y)h^T \\ &\quad + 4\eta^2 h(W h - y)^T (W h - y)h^T \\ W_{new}^T W_{new} &= Q - 2\eta(h h^T W^T W - h y^T W) - 2\eta(W^T W h h^T - W^T y h^T) \\ &\quad + 4\eta^2 h(h^T W^T W h - h^T W^T y - y^T W h + y^T y)h^T \\ W_{new}^T W_{new} &= Q - 2\eta(h h^T Q - h(W^T y)^T) - 2\eta(Q h h^T - (W^T y)h^T) \\ &\quad + 4\eta^2 h(h^T Q h - h^T (W^T y) - (W^T y)^T h + y^T y)h^T \\ W_{new}^T W_{new} &= Q - 2\eta h(h^T Q - (W^T y)^T) - 2\eta(Q h - W^T y)h^T \end{aligned}$$

$$\begin{aligned}
& +4\eta^2 h(h^T Q h - 2h^T W^T y + y^T y)h^T \\
W_{new}^T W_{new} &= Q - \eta h \underbrace{(2(Qh - W^T y))^T}_{\nabla_h} - \eta \underbrace{(2(Qh - W^T y))}_{\nabla_h} h^T \\
& +4\eta^2 h \underbrace{(h^T (Qh - 2W^T y) + y^T y)}_L h^T
\end{aligned}$$

where we see that the last term uses the expression of L from Eq. 1 on page 6 and the first two terms uses the expression of ∇_h from Eq. 6: $\nabla_h = 2(Qh - U^T(V^T y)) = 2(Qh - W^T y)$. Thus we have shown that

$$\begin{aligned}
W_{new}^T W_{new} &= Q - \eta h \nabla_h^T - 2\eta \nabla_h h^T + 4\eta^2 h L h^T \\
&= Q - \eta (h \nabla_h^T + \nabla_h h^T) + (4\eta^2 L) h h^T
\end{aligned}$$

which is the update Q_{new} that we gave in Eq. 8 above.

3.5 Putting it all together: detailed online update algorithm and expected benefits

We have seen that we can efficiently compute cost L , gradient with respect to h (to be later backpropagated further) as well as updating U and V and performing the book-keeping for U^{-T} and Q . Here we put everything together. The parameters of the output layer that we will learn are V, U and implicitly represent W as $W = VU$. We first need to initialize these parameter matrices, as well as bookkeeping matrices Q and U^{-T} in a consistent way, as explained in Algo. 1. We then iterate over the following:

- pick a next *input,target* example x, y (where y is K -sparse and uses an appropriate sparse representation)
- perform forward propagation through all layers of the network up to the last hidden layer, to compute last hidden layer representation $h = h(x)$, that should include a constant 1 first element.
- execute Algo. 2, that we put together from the equations derived above, and that will: compute the associated squared error loss L , perform an implicit gradient update step on W by correspondingly updating V and U in a computationally efficient manner, update bookkeeping matrices Q and U^{-T} accordingly, and compute and return the gradient of the loss with respect to the last hidden layer ∇_h
- having ∇_h , further backpropagate the gradients upstream, and use them to update the parameters of all other layers

Having $K \ll d \ll D$ we see that the update algorithm we developed requires $O(d^2)$ operations, whereas the standard approach required $O(Dd)$ operations. If we take $K \approx d$, we may state more precisely that the proposed algorithm, for computing the loss and the gradient updates will require roughly $12d^2$ operations whereas the standard approach required roughly $3Dd$ operations. So overall the proposed algorithm

change corresponds to a computational speedup by a factor of $\frac{D}{Ad}$. For $D = 200\,000$ and $d = 500$ the expected speedup is thus **100**. Note that the advantage is not only in *computational* complexity, but also in *memory access*. For each example, the standard approach needs to access and change all $D \times d$ elements of matrix W , whereas the proposed approach only accesses the much smaller number $K \times d$ elements of V as well as the three $d \times d$ matrices U , U^{-T} , and Q . So overall we have a **substantially faster algorithm whose complexity is independent of D** , which, while doing so *implicitly*, will nevertheless perform the *exact same* gradient update as the standard $O(Dd)$ approach. We want to emphasize here that this approach is entirely different from simply chaining 2 linear layers U and V and performing ordinary gradient descent updates on these: this would result in the same prohibitive computational complexity as the standard approach, and such ordinary separate gradient updates to U and V would *not* be equivalent to the ordinary gradient update to $W = VU$.

Algorithm 1 Initialization of output layer parameters V, U and bookkeeping matrices Q, U^{-T}

- we can initialize $D \times d$ matrix V randomly as we would have initialized W so that we initially have $V = W$.
Alternatively we can initialize V to 0 (there won't be symmetry breaking issues with having W initially be 0 provided the other layers are initialized randomly, since varying inputs and targets will naturally break symmetry for the output layer)
 - initialize $Q \leftarrow V^T V$ (or more cheaply initialize $Q \leftarrow 0$ if we have initialized V to 0).
 - we initialize U to the identity: $U \leftarrow \mathbf{I}_d$ so that, trivially, we initially have $VU = W$.
 - initialize $U^{-T} \leftarrow \mathbf{I}_d$
-

3.6 Minibatch version of the algorithm for squared error

The algorithm we derived for online gradient is relatively straightforward to extend to the case of minibatches containing m examples. We initialize parameters as in the online case following Algo. 1 and apply the same training procedure outlined in Section. 3.5, but now using minibatches containing m examples, rather than a single example vector. The corresponding update and gradient computation is given in Algorithm 3 which follows equivalent steps to the online version of Algorithm 2, but using matrices with m columns in place of single column vectors. For example step 3 which in the online algorithm was $\nabla_h = 2(\hat{h} - \hat{y})$ using d -dimensional vectors becomes in the minibatch version $\nabla_H = 2(\hat{H} - \hat{Y})$ using $d \times m$ matrices instead.

Note that in the minibatch version, in step 6, we update U^{-T} based on the Woodbury equation, which generalizes the Sherman-Morrison formula for $m > 1$ and involves inverting an $m \times m$ matrix, an $O(m^3)$ operation. But depending on the size of the minibatch m , it may become more efficient to solve the corresponding linear equations for each minibatch from scratch every time, rather than inverting that $m \times m$ matrix. In which case we won't need to maintain an U^{-T} at all. Or in cases of minibatches containing more than d examples, it may even become more efficient to invert U from scratch every time.

Algorithm 2 Efficient computation of cost L , gradient ∇h , and update to parameters U and V for squared error, in the online case

Inputs (besides above parameters V, U, Q, U^{-T}):

- $h \in \mathbb{R}^d$ hidden representation vector for one example $h \in \mathbb{R}^d$
- $y \in \mathbb{R}^D$ associated K -sparse target vector stored using a sparse representation (indices and values of non-zero elements)
- $\eta \in \mathbb{R}^+$ learning rate for the update

Outputs:

- $L \in \mathbb{R}$ the squared error loss for this example
- updated parameters and bookkeeping matrices $U_{new}, V_{new}, Q_{new}, U_{new}^{-T}$
- $\nabla_h \in \mathbb{R}^d$ the gradient of the loss with respect to h , to further backpropagate upstream.

Algorithm:

Step #	Operation	Computational complexity	Approximate number of elementary operations (multiply-adds)
1:	$\hat{h} = Qh$	$O(d^2)$	d^2
2:	$\hat{y} = U^T(V^T y)$	$O(Kd + d^2)$	$Kd + d^2$
3:	$\nabla_h = 2(\hat{h} - \hat{y})$	$O(d)$	d
4:	$L = h^T \hat{h} - 2h^T \hat{y} + y^T y$	$O(2d + K)$	$2d + K + 1$
5:	$U \leftarrow U - 2\eta(Uh)h^T$	$O(d^2)$	$2d^2 + d$
6:	$U^{-T} \leftarrow U^{-T} + \frac{2\eta}{1-2\eta\ h\ ^2}(U^{-T}h)h^T$ [from Sherman-Morrison formula]	$O(d^2)$	$2d^2 + 2d + 3$
7:	$V \leftarrow V + 2\eta y(U^{-T}h)^T$ where we must use the freshly updated U^{-T} resulting from step 6)	$O(d^2 + Kd)$	$d^2 + K + Kd$
8:	$Q \leftarrow Q - \eta(h\nabla_h^T + \nabla_h h^T) + (4\eta^2 L)hh^T$	$O(d^2)$	$4 + 2d + 3d^2$
	Altogether:	$O(d^2)$ provided $K < d \ll D$	$\approx 12d^2$ elementary operations

In step 9, the update Q_{new} for Q corresponds to the implicit weight update $W_{new} \leftarrow W - 2\eta(WH - Y)H^T$ as we now prove:

We will use the following precomputed quantities: $Q = W^T W$, $\hat{H} = QH$ and $\hat{Y} = W^T Y = U^T(V^T Y)$ and $\nabla_H = 2(\hat{H} - \hat{Y})$.

$$\begin{aligned}
Q_{new} &= W_{new}^T W_{new} \\
&= (W - 2\eta(WH - Y)H^T)^T (W - 2\eta(WH - Y)H^T) \\
&= W^T W - 2\eta H(WH - Y)^T W - 2\eta W^T(WH - Y)H^T \\
&\quad + 4\eta^2 H(WH - Y)^T(WH - Y)H^T \\
&= Q - 2\eta(HH^T W^T W - HY^T W) - 2\eta(W^T W H H^T - W^T Y H^T) \\
&\quad + 4\eta^2 H(H^T W^T W H - H^T W^T Y - Y^T W H + Y^T Y)H^T \\
&= Q - 2\eta(HH^T Q - H(W^T Y)^T) - 2\eta(QH H^T - (W^T Y)H^T) \\
&\quad + 4\eta^2 H(H^T Q H - H^T(W^T Y) - (W^T Y)^T H + Y^T Y)H^T \\
&= Q - 2\eta(H\hat{H}^T - H\hat{Y}^T + \hat{H}H^T - \hat{Y}H^T) \\
&\quad + 4\eta^2 H(H^T \hat{H} - H^T \hat{Y} - \hat{Y}^T H + Y^T Y)H^T \\
&= Q - 2\eta(H(\hat{H} - \hat{Y})^T + (\hat{H} - \hat{Y})H^T) + 4\eta^2 H(H^T(\hat{H} - \hat{Y}) - \hat{Y}^T H + Y^T Y)H^T \\
&= Q - \eta(H(2(\hat{H} - \hat{Y}))^T + (2(\hat{H} - \hat{Y}))H^T) + 4\eta^2 H(H^T(\hat{H} - \hat{Y}) - \hat{Y}^T H + Y^T Y)H^T \\
&= Q - \eta(H\nabla_H^T + \nabla_H H^T) + 4\eta^2 H \underbrace{(H^T \hat{Z} - \hat{Y}^T H + Y^T Y)}_M H^T
\end{aligned}$$

which is the update of Q we use in in step 8 of Algorithm on the previous page.

Algorithm 3 Minibatch version of the update algorithm for squared error

Inputs (besides above parameters V, U, Q, U^{-T}):

- parameters and bookkeeping matrices: U, V, Q, U^{-T}
- H : a $d \times m$ matrix whose m columns contain the last hidden layer representation vectors for m example (with an appended constant 1 element to account for an output bias).
- Y : a $D \times m$ sparse target matrix. Each of its m columns is the K -sparse target vector associated to one example of the minibatch, stored using a sparse representation (indices and values of non-zero elements).
- $\eta \in \mathbb{R}^+$ learning rate for the update

Updates:

- parameters and bookkeeping matrices: U, V, Q, U^{-T}

Outputs:

- $L \in \mathbb{R}$ the sum of squared error losses for the m examples of the minibatch
- ∇_H a $d \times m$ matrix whose m columns contain the gradient of the loss with respect to H , to further backpropagate upstream.

Algorithm:

Step #	Operation	Computation complexity	Approximate number of elementary operations (multiply-adds)
1:	$\hat{H} = QH$	$O(md^2)$	md^2
2:	$\hat{Y} = U^T(V^T Y)$	$O(mKd + md^2)$	$mKd + md^2$
3:	$\nabla_H = 2(\hat{H} - \hat{Y})$	$O(md)$	md
4a:	$M = H^T \hat{H} - (\hat{Y}^T H + H^T \hat{Y}) + Y^T Y$	$O(m^2d + m^2K)$	$2m^2d + m^2K$
4b:	$L = \text{Tr}(M)$	$O(m)$	m
5:	$U \leftarrow U - 2\eta(UH)H^T$	$O(md^2)$	$2md^2 + md$
6:	$U^{-T} \leftarrow U^{-T} - (U^{-T}H) \left((H^T H - \frac{1}{2\eta} \mathbf{I}_m)^{-1} H^T \right)$ [from Woodbury identity]	$O(m^2d + m^3 + md^2)$	$2md^2 + m + \frac{2}{3}m^3 + m^2d$ (we count $\frac{2}{3}m^3$ operations for inversion of a $m \times m$ matrix)
7:	$V \leftarrow V + 2\eta Y(U^{-T}H)^T$ where we must use the freshly updated U^{-T} resulting from step 6)	$O(md^2 + mKd)$	$md^2 + mK + mKd$
8:	$Q \leftarrow Q - \eta(H\nabla_H^T + \nabla_H H^T) + 4\eta^2(HM)H^T$	$O(md^2 + m^2d)$	$m^2d + 3md^2 + 2d^2$
	Altogether:	$O(md^2)$ provided $K < m < d \ll D$.	$\approx 10md^2 + 3m^2d + m^3$ elementary operations when $K = 1$

Note that if we chose $m > d$ we will not perform step 7 based on the Woodbury identity, which would be wasteful, but instead directly recompute the inverse of U_{new} in $O(d^3)$. The overall complexity remains $O(md^2)$ in this case also.

4 Generalizing to a broader family of loss functions

Let $o = Wh$ the linear activations computed at the output layer. The approach that we detailed for linear output and squared error can be extended to a more general family of loss functions: basically any loss function ℓ that can be expressed using only the o_c associated to non-zero y_c together with $q = \|o\|^2 = \sum_j o_j^2$ the squared norm of the whole output vector, and optionally $s = \text{sum}(o) = \sum_j o_j$ which we will see that we can both compute cheaply. We call this family of loss functions the *spherical family of loss functions* or in short *spherical losses*, defined more formally as the family of losses that can be expressed as:

$$L = \ell(\|o\|^2, \text{sum}(o), \mathcal{K}, o_{\mathcal{K}}, y_{\mathcal{K}})$$

where \mathcal{K} denotes the vector of indices of y of cardinality at most $K \ll D$ that is associated to non-zero elements of y in a sparse representation of y ; $y_{\mathcal{K}}$ is the corresponding vector of values of y at positions \mathcal{K} , i.e. $y_{\mathcal{K}} = (y_{(\mathcal{K}_1)}, \dots, y_{(\mathcal{K}_{|\mathcal{K}|})})^T$; similarly $o_{\mathcal{K}}$ is the vector of values of linear activation o at positions \mathcal{K} , i.e. $o_{\mathcal{K}} = (o_{(\mathcal{K}_1)}, \dots, o_{(\mathcal{K}_{|\mathcal{K}|})})^T$.

Note that the squared error loss belongs to this family as

$$\begin{aligned} \ell_{\text{squared}} &= \sum_{j=1}^D (o_j - y_j)^2 \\ &= \sum_{j=1}^D o_j^2 - 2o_j y_j + y_j^2 \\ &= \left(\sum_{j=1}^D o_j^2 \right) - 2 \left(\sum_{j=1}^D o_j y_j \right) + \left(\sum_{j=1}^D y_j^2 \right) \\ &= \|o\|^2 - 2 \left(\sum_{j \in \mathcal{K}} o_j y_j \right) + \left(\sum_{j \in \mathcal{K}} y_j^2 \right) \quad \text{since for } j \notin \mathcal{K} \text{ we have } y_j = 0 \\ &= \|o\|^2 - 2o_{\mathcal{K}}^T y_{\mathcal{K}} + \|y_{\mathcal{K}}\|^2 \\ &= \ell_{\text{squared}}(\|o\|^2, \text{sum}(o), \mathcal{K}, o_{\mathcal{K}}, y_{\mathcal{K}}) \end{aligned}$$

where ℓ_{squared} in particular doesn't use $\text{sum}(o)$.

The spherical family of loss functions does not include the standard log of softmax, but it includes possible alternatives, such as the *spherical softmax* and *Taylor-softmax* that we will introduce in a later section. Let us detail the steps for computing such a spherical loss from last hidden layer representation h :

- $o = Wh$
- $q = \|o\|^2 = \sum_i o_i^2$
- $s = \text{sum}(o) = \sum_i o_i$
- $L = \ell(q, s, \mathcal{K}, o_{\mathcal{K}}, y_{\mathcal{K}})$

The gradient of the loss may be backpropagated and the parameters updated in the *usual naive* way with the following steps:

- compute scalars $\frac{\partial \ell}{\partial q}(q, s, \mathcal{K}, o_{\mathcal{K}}, y_{\mathcal{K}})$ and $\frac{\partial \ell}{\partial s}(q, s, \mathcal{K}, o_{\mathcal{K}}, y_{\mathcal{K}})$ as well as K -dimensional gradient vector $\frac{\partial \ell}{\partial o_{\mathcal{K}}}(q, s, \mathcal{K}, o_{\mathcal{K}}, y_{\mathcal{K}})$
- clear D -dimensional gradient vector $\nabla_o \leftarrow 0$
- update $(\nabla_o)_{\mathcal{K}} \leftarrow \frac{\partial \ell}{\partial o_{\mathcal{K}}}$
- update $\nabla_o \leftarrow \nabla_o + \frac{\partial \ell}{\partial q} \underbrace{\frac{\partial q}{\partial o}}_{2o}$
- update $\nabla_o \leftarrow \nabla_o + \frac{\partial \ell}{\partial s} \underbrace{\frac{\partial s}{\partial o}}_{\frac{1_D}{2o}}$
- backpropagate $\nabla_h = W^T \nabla_o$
- update $W \leftarrow W - \eta \nabla_o h^T$ where η is a scalar learning rate.

Here again, as in the squared error case, we see that the computation of o in the forward pass and backpropagation of the gradient to ∇_h would both require multiplication by the $D \times d$ matrix W , and that the update to W will generally be a non-sparse rank-1 update that requires modifying all its Dd elements. Each of these three operations have a $O(Dd)$ complexity.

We will now follow the same logical steps as in the simpler squared error case to derive an efficient algorithm for the spherical loss family.

4.1 Efficient computation of the loss

Let us name the formal parameters of ℓ more clearly as follows:

$$\ell(q, s, \mathcal{K}, \mathbf{a}, \mathbf{t})$$

where q and s are scalars that will receive $\|o\|^2$ and $\text{sum}(o)$ respectively; \mathcal{K} is a vector that will contain the list of at most K indices that correspond to non-zero elements of sparse y ; $\mathbf{a} = o_{\mathcal{K}}$ and $\mathbf{t} = y_{\mathcal{K}}$.

4.1.1 Computing $q = \|o\|^2$

$$\begin{aligned} q = \|o\|^2 &= \|\underbrace{Wh}_{O(Dd)}\|^2 \\ &= (Wh)^T (Wh) \\ &= h^T W^T W h \\ &= h^T \underbrace{(Qh)}_{O(d^2)} \end{aligned} \tag{9}$$

supposing we have maintained an up-to date $Q = W^T W$.

Derivative:

$$\frac{\partial q}{\partial o} = 2o$$

4.1.2 Computing $s = \text{sum}(o)$

$$\begin{aligned}
s = \text{sum}(o) &= \text{sum}(\overbrace{Wh}^{O(Dd)}) \\
&= \sum_{i=1}^D \left(\sum_{j=1}^d h_j W_{ij} \right)_i \\
&= \sum_{i=1}^D \sum_{j=1}^d h_j W_{ij} \\
&= \sum_{j=1}^d \left(h_j \sum_{i=1}^D W_{ij} \right) \\
&= \sum_{j=1}^d h_j \underbrace{\text{sum}(W_j)}_{\bar{w}_j} \\
&= \bar{w}^T h \\
&= h^T \bar{w}
\end{aligned} \tag{10}$$

This is an $O(d)$ operation, provided we have maintained an up-to-date vector $\bar{w} = (\text{sum}(W_1), \dots, \text{sum}(W_d)) = W^T \mathbf{1}_D$.

$$\frac{\partial s}{\partial o} = \mathbf{1}_D$$

4.1.3 Computing specific o_k

We will also need to compute the specific o_k for the few $k \in \mathcal{K}$.

$$\begin{aligned}
o_k &= (Wh)_k \\
&= h^T W_{k\bullet}
\end{aligned}$$

which gives

$$\begin{aligned}
\mathbf{a} = o_{\mathcal{K}} &= (o_{(\mathcal{K}_1)}, \dots, o_{(\mathcal{K}_{|\mathcal{K}|})})^T \\
&= (h^T W_{\mathcal{K}_1\bullet}, \dots, h^T W_{\mathcal{K}_{|\mathcal{K}|}\bullet})^T
\end{aligned} \tag{11}$$

we then have all we need to pass to loss function ℓ to compute the associated loss

$$L = \ell(q, s, \mathcal{K}, o_{\mathcal{K}}, y_{\mathcal{K}}) = \ell(q, s, \mathcal{K}, \mathbf{a}, \mathbf{t}) \tag{12}$$

4.1.4 Corresponding equations for the minibatch case

In the minibatch case, rather than having the hidden representation of a single example as a vector h we suppose we receive m hidden representations in the m columns of a $d \times m$ matrix H . The associated sparse target is $D \times m$ matrix Y whose m columns contain each at most K non-zero elements. Y will be stored using sparse representation (\mathcal{K}, T) where \mathcal{K} is now a $K \times m$ matrix of indices and T is a $K \times m$ matrix containing the corresponding values of Y such that $T_{kj} = Y_{\mathcal{K}_{kj}, j}$ for $k \in \{1, \dots, K\}$ and $j \in \{1, \dots, m\}$.

The above equations given for the online case, can easily be adapted to the minibatch case as follows:

Let $O = WH$ the $D \times m$ matrix of linear outputs whose j^{th} column will contain the output vector of the j^{th} example of the minibatch. The specific outputs associated to non-zero target values in Y (whose indexes are in \mathcal{K}) will be collected in $K \times m$ matrix A (the minibatch version of vector \mathbf{a} of Equation 11 such that

$$A_{kj} = O_{\mathcal{K}_{kj}, j} = (H_j)^T W_{\mathcal{K}_{kj}, \bullet} \quad (13)$$

Adapting Equation 9 to the minibatch case, the squared norm of the m output vectors is obtained in m -dimensional vector \mathbf{q} as

$$\mathbf{q} = \text{diag}(H^T \underbrace{QH}_{\hat{H}}) \quad (14)$$

Adapting Equation 10 to the minibatch case, the sum of each of the m output vectors is obtained in m -dimensional vector \mathbf{s} as

$$\mathbf{s} = H^T \bar{w} \quad (15)$$

Adapting Equation 12 the corresponding vector of m individual losses for the m examples of the minibatch is

$$\vec{L} = [\ell(\mathbf{q}_j, \mathbf{s}_j, \mathcal{K}_j, A_j, T_j)]_{j=1 \dots m} \quad (16)$$

and the total loss for the minibatch is

$$L = \text{sum}(\vec{L}) \quad (17)$$

4.2 Gradient of loss L with respect to h

Online case:

To backpropagate the gradients through the network, we first need the gradients with respect to linear activations o : $\nabla_o = \frac{\partial L}{\partial o}$.

There will be three types of contributions to this gradient: contribution due to q , contribution due to s , and contribution due to *direct* influence on the loss of the o_k for $k \in \mathcal{K}$.

$$\nabla_o = \frac{\partial L}{\partial o} = \frac{\partial \ell}{\partial q} \frac{\partial q}{\partial o} + \frac{\partial \ell}{\partial s} \frac{\partial s}{\partial o} + \sum_{k=1}^K \frac{\partial \ell}{\partial \mathbf{a}_k} \frac{\partial \mathbf{a}_k}{\partial o}$$

We have $\frac{\partial q}{\partial o} = 2o$, $\frac{\partial s}{\partial o} = \mathbf{1}_D$ and $\frac{\partial \mathbf{a}_k}{\partial o} = \text{onehot}_D(\mathcal{K}_k)$ because $\mathbf{a}_k = o_{\mathcal{K}_k}$ so this becomes

$$\begin{aligned} \nabla_o &= 2o \frac{\partial \ell}{\partial q} + \mathbf{1}_D \frac{\partial \ell}{\partial s} + \sum_{k=1}^K \frac{\partial \ell}{\partial \mathbf{a}_k} \text{onehot}_D(\mathcal{K}_k) \\ &= 2o \frac{\partial \ell}{\partial q} + \mathbf{1}_D \frac{\partial \ell}{\partial s} + \dot{y} \end{aligned} \tag{18}$$

where we have defined vector $\dot{y} = \sum_{k=1}^K \frac{\partial \ell}{\partial \mathbf{a}_k} \text{onehot}_D(\mathcal{K}_k)$ as a sparse vector, having value at position \mathbf{k}_j equal $\frac{\partial \ell}{\partial \mathbf{a}_j}$. It will, like y , be stored in K - *sparse* representation, with the indexes given by \mathbf{k} and the corresponding values in $\frac{\partial \ell}{\partial \mathbf{a}_j}$.

Gradient with respect to h :

$$\begin{aligned} \nabla_h &= \frac{\partial o}{\partial h} \frac{\partial L}{\partial o} \\ &= W^T \nabla_o \\ &= W^T \left(2o \frac{\partial \ell}{\partial q} + \mathbf{1}_D \frac{\partial \ell}{\partial s} + \dot{y} \right) \\ &= 2W^T o \frac{\partial \ell}{\partial q} + W^T \mathbf{1}_D \frac{\partial \ell}{\partial s} + W^T \dot{y} \\ &= 2W^T W h \frac{\partial \ell}{\partial q} + \bar{w} \frac{\partial \ell}{\partial s} + W^T \dot{y} \\ &= 2Qh \frac{\partial \ell}{\partial q} + \bar{w} \frac{\partial \ell}{\partial s} + W^T \sum_{k=1}^K \frac{\partial \ell}{\partial \mathbf{a}_k} \text{onehot}_D(\mathcal{K}_k) \\ &= 2Qh \frac{\partial \ell}{\partial q} + \bar{w} \frac{\partial \ell}{\partial s} + \sum_{k=1}^K \frac{\partial \ell}{\partial \mathbf{a}_k} W^T \text{onehot}_D(\mathcal{K}_k) \\ &= 2Qh \frac{\partial \ell}{\partial q} + \bar{w} \frac{\partial \ell}{\partial s} + \sum_{k=1}^K \frac{\partial \ell}{\partial \mathbf{a}_k} W_{\mathcal{K}_k \bullet} \end{aligned}$$

Minibatch case:

We now consider a minibatch of m examples whose corresponding linear outputs are in a $D \times m$ matrix $O = WH$. Let us also denote the vectors of gradients of the loss with respect to \mathbf{q} and \mathbf{s} as:

$$\begin{aligned}\nabla_q &= \left[\frac{\partial \ell}{\partial q}(\mathbf{q}_j, \mathbf{s}_j, \mathcal{K}_j, A_j, T_j) \right]_{j=1 \dots m} \\ \nabla_s &= \left[\frac{\partial \ell}{\partial s}(\mathbf{q}_j, \mathbf{s}_j, \mathcal{K}_j, A_j, T_j) \right]_{j=1 \dots m}\end{aligned}$$

Let us also define

$$\nabla_A = \left[\frac{\partial \ell}{\partial \mathbf{a}_k}(\mathbf{q}_j, \mathbf{s}_j, \mathcal{K}_j, A_j, T_j) \right]_{k=1 \dots K, j=1 \dots m}$$

and \mathring{Y} as the sparse $D \times m$ whose column j is defined as

$$\begin{aligned}\mathring{Y}_j &= \sum_{k=1}^K \frac{\partial \ell}{\partial \mathbf{a}_k}(\mathbf{q}_j, \mathbf{s}_j, \mathcal{K}_j, A_j, T_j) \text{onehot}_D(\mathcal{K}_{kj}) \\ &= \sum_{k=1}^K (\nabla_A)_{kj} \text{onehot}_D(\mathcal{K}_{kj})\end{aligned}$$

which may be summarize as $\mathring{Y}_{\mathcal{K}_j} = (\nabla_A)_j$

Equation 18 then becomes in the minibatch case:

$$\nabla_{O_j} = 2O_j (\nabla_q)_j + \mathbf{1}_D (\nabla_s)_j + \mathring{Y}_j$$

or in matrix form

$$\nabla_O = 2O \text{diag}(\nabla_q) + \mathbf{1}_D \nabla_s^T + \mathring{Y} \quad (19)$$

and the gradient with respect to H is:

$$\begin{aligned}\nabla_H &= \frac{\partial L}{\partial H} \\ &= \frac{\partial O}{\partial H} \frac{\partial L}{\partial O} \\ &= W^T \nabla_O \quad (20)\end{aligned}$$

$$\begin{aligned}&= W^T \left(2O \text{diag}(\nabla_q) + \mathbf{1}_D \nabla_s^T + \mathring{Y} \right) \\ &= 2W^T O \text{diag}(\nabla_q) + W^T \mathbf{1}_D \nabla_s^T + W^T \mathring{Y} \\ &= 2W^T W H \text{diag}(\nabla_q) + \bar{w} \nabla_s^T + W^T \mathring{Y} \\ &= 2QH \text{diag}(\nabla_q) + \underbrace{\bar{w} \nabla_s^T + W^T \mathring{Y}}_{\hat{Z}} \quad (21)\end{aligned}$$

where we define the $d \times m$ matrix \hat{Z} as

$$\hat{Z} = \bar{w} \nabla_s^T + W^T \mathring{Y} \quad (22)$$

4.3 Standard naive gradient update of parameters W

The gradient of the loss with respect to output layer weight matrix W is

$$\begin{aligned}
\frac{\partial L}{\partial W} &= \frac{\partial L}{\partial O} \frac{\partial O}{\partial W} \\
&= \nabla_O H^T \\
&= \left(2O \text{diag}(\nabla_q) + \mathbf{1}_D \nabla_s^T + \dot{Y} \right) H^T \\
&= \left(2WH \text{diag}(\nabla_q) + \mathbf{1}_D \nabla_s^T + \dot{Y} \right) H^T
\end{aligned}$$

And the corresponding gradient descent update to W would thus be

$$W_{new} = W - \eta \left(2WH \text{diag}(\nabla_q) + \mathbf{1}_D \nabla_s^T + \dot{Y} \right) H^T \quad (23)$$

where η is a positive learning rate.

Computed in this manner, this induces a prohibitive $O(mDd)$ computational complexity, first to compute WH , and then to update all the Dd elements of W . Note that all $D \times d$ elements of W must be accessed during this update. On the surface this seems hopeless. But we will see in the next section how we can achieve the *exact* same update of W in $O(md^2)$.

4.4 Efficient gradient update of parameters using a factored representation of W

First note that the update of W given in equation 23 can be decomposed in 3 consecutive updates:

$$\begin{aligned}
\text{a) } W &\leftarrow W - 2\eta(WH) \text{diag}(\nabla_q) H^T \\
\text{b) } W &\leftarrow W - \eta \mathbf{1}_D \nabla_s^T H^T \\
\text{c) } W &\leftarrow W - \eta \dot{Y} H^T
\end{aligned}$$

In doing this we haven't yet changed anything to the $O(mDd)$ complexity of this update. Note that update a) can also be seen as $W \leftarrow W (\mathbf{I} - 2\eta H \text{diag}(\nabla_q) H^T)$.

The trick now is to represent W implicitly as³:

$$\underbrace{W}_{D \times d} = \underbrace{V}_{D \times d} \underbrace{U}_{d \times d} + \mathbf{1}_D \omega^T \quad (24)$$

where ω is a d -dimensional vector. In this case the following updates to V, U, ω respectively will implicitly update the implicit W in the exact same way as the above 3 updates:

³Note that we never actually *factorize* an arbitrary pre-existing W , which would be prohibitive as W is huge. We will no longer store or update a W , but only V, U, ω which implicitly represent W .

$$\begin{aligned} \text{a) } U_{new} &= U (\mathbf{I} - 2\eta H \text{diag}(\nabla_q) H^T) \\ &= U - 2\eta U H \text{diag}(\nabla_q) H^T \end{aligned} \quad (25)$$

$$\begin{aligned} \text{b) } \omega_{new} &= (\mathbf{I} - 2\eta H \text{diag}(\nabla_q) H^T)^T \omega - \eta H \nabla_s \\ &= \omega - 2\eta H \text{diag}(\nabla_q) H^T \omega - \eta H \nabla_s \\ &= \omega - \eta H (2 \text{diag}(\nabla_q) H^T \omega + \nabla_s) \end{aligned} \quad (26)$$

$$\text{c) } V_{new} = V - \eta \dot{Y} (U_{new}^{-T} H)^T \quad (27)$$

But, with this formulation, provided we keep an up-to-date U^{-T} (which we will see we can do cheaply using the Woodbury identity), the whole update to V, U, ω is now $O(md^2)$ rather than the equivalent naive $O(mDd)$ update of Eq. 23 to an explicit W .

Indeed, step a) and b) involve only multiplications between matrices of dimensions $d \times m$ and $d \times d$ (matrices H and U). As for step c) it involves an $O(md^2)$ multiplication of U^{-T} by H , followed by a **sparse update** of V . Since \dot{Y} is an extremely sparse $D \times m$ matrix whose m columns each contain at most K non-zero elements, update c) will touch at most Km rows of V , yielding an $O(Kmd)$ operation. This is to be contrasted with the standard, equivalent but naive update of Eq. 23 to an explicit W , which requires accessing and modifying all $D \times d$ elements of W for every update and yields an overall $O(mDd)$ computational complexity.

Proof that this sequence of updates yields the update of W given above:

$$\begin{aligned} & V_{new} U_{new} + \mathbf{1}_D \omega_{new}^T \\ &= \left(V - \eta \dot{Y} (U_{new}^{-T} H)^T \right) U_{new} + \mathbf{1}_D (\omega - \eta H (2 \text{diag}(\nabla_q) H^T \omega + \nabla_s))^T \\ &= \left(V - \eta \dot{Y} H^T U_{new}^{-1} \right) U_{new} + \mathbf{1}_D (\omega^T - \eta (2 \text{diag}(\nabla_q) H^T \omega + \nabla_s)^T H^T) \\ &= V U_{new} - \eta \dot{Y} H^T U_{new}^{-1} U_{new} + \mathbf{1}_D \omega^T - \eta \mathbf{1}_D (2 \text{diag}(\nabla_q) H^T \omega + \nabla_s)^T H^T \\ &= V U_{new} - \eta \dot{Y} H^T + \mathbf{1}_D \omega^T - \eta \mathbf{1}_D (2 \text{diag}(\nabla_q) H^T \omega + \nabla_s)^T H^T \\ &= V (U - 2\eta U H \text{diag}(\nabla_q) H^T) - \eta \dot{Y} H^T + \mathbf{1}_D \omega^T - \eta \mathbf{1}_D (2 \text{diag}(\nabla_q) H^T \omega + \nabla_s)^T H^T \\ &= V U - 2\eta V U H \text{diag}(\nabla_q) H^T - \eta \dot{Y} H^T + \mathbf{1}_D \omega^T - \eta \mathbf{1}_D (2 \omega^T H \text{diag}(\nabla_q) + \nabla_s^T) H^T \\ &= (V U + \mathbf{1}_D \omega^T) - 2\eta V U H \text{diag}(\nabla_q) H^T - \eta \dot{Y} H^T - \eta \mathbf{1}_D (2 \omega^T H \text{diag}(\nabla_q) + \nabla_s^T) H^T \\ &= W - 2\eta V U H \text{diag}(\nabla_q) H^T - \eta \dot{Y} H^T - 2\eta \mathbf{1}_D \omega^T H \text{diag}(\nabla_q) H^T - \eta \mathbf{1}_D \nabla_s^T H^T \\ &= W - 2\eta V U H \text{diag}(\nabla_q) H^T - 2\eta \mathbf{1}_D \omega^T H \text{diag}(\nabla_q) H^T - \eta \mathbf{1}_D \nabla_s^T H^T - \eta \dot{Y} H^T \\ &= W - 2\eta (V U H \text{diag}(\nabla_q) H^T + \mathbf{1}_D \omega^T H \text{diag}(\nabla_q) H^T) - \eta \mathbf{1}_D \nabla_s^T H^T - \eta \dot{Y} H^T \\ &= W - 2\eta (V U + \mathbf{1}_D \omega^T) H \text{diag}(\nabla_q) H^T - \eta \mathbf{1}_D \nabla_s^T H^T - \eta \dot{Y} H^T \\ &= W - 2\eta W H \text{diag}(\nabla_q) H^T - \eta \mathbf{1}_D \nabla_s^T H^T - \eta \dot{Y} H^T \end{aligned}$$

$$\begin{aligned}
&= W - \eta \left(2WH \text{diag}(\nabla_q) + \mathbf{1}_D \nabla_s^T + \hat{Y}^\circ \right) H^T \\
&= W_{new}
\end{aligned}$$

4.5 Adapting the computation of loss L and gradient ∇_H to the factorized representation

Let us now adapt the computation of loss L and gradient ∇_H now that we no longer have an explicit W but rather store it implicitly as $W = VU + \mathbf{1}_D \omega^T$.

4.5.1 Loss L

Computing the total loss L over a minibatch implies computing $L = \text{sum}(\vec{L}) = \text{sum} \left([\ell(\mathbf{q}_j, \mathbf{s}_j, \mathcal{K}_j, A_j, T_j)]_{j=1 \dots m} \right)$ as previously seen in Eq. 16 and Eq. 17. Index matrix \mathcal{K} and associated target matrix T are the same as before. Vectors \mathbf{q} and \mathbf{s} can be computed cheaply as previously using Eq. 14 and 15 provided we have kept an up-to-date Q and \bar{w} (we shall see how to update them effectively in the next section). So to be able to compute loss L using this factored representation of W it remains only to adapt the computation of $K \times m$ matrix A . This matrix was defined in Eq. 13 as $A_{kj} = O_{\mathcal{K}_{kj}, j} = (H_j)^T W_{\mathcal{K}_{kj}, \bullet}$. Replacing W by its factored expression we can write

$$\begin{aligned}
A_{kj} &= (H_j)^T (VU + \mathbf{1}_D \omega^T)_{\mathcal{K}_{kj}, \bullet} \\
&= (H_j)^T (VU)_{\mathcal{K}_{kj}, \bullet} + (H_j)^T (\mathbf{1}_D \omega^T)_{\mathcal{K}_{kj}, \bullet} \\
&= (H_j)^T (VU)_{\mathcal{K}_{kj}, \bullet} + (H_j)^T \omega \\
&= (H_j)^T ((VU)^T)_{\mathcal{K}_{kj}} + (H_j)^T \omega \\
&= (H_j)^T (U^T V^T)_{\mathcal{K}_{kj}} + (H_j)^T \omega \\
&= (H_j)^T U^T (V^T)_{\mathcal{K}_{kj}} + (H_j)^T \omega \\
&= (UH_j)^T (V^T)_{\mathcal{K}_{kj}} + (H_j)^T \omega \\
&= ((UH)_j)^T V_{\mathcal{K}_{kj}, \bullet} + (H_j)^T \omega \\
&= (\underbrace{(UH)_j}_{\tilde{H}})^T V_{\mathcal{K}_{kj}, \bullet} + (\underbrace{H^T \omega}_j)
\end{aligned}$$

In summary, having computed

$$\tilde{H} = UH \tag{28}$$

and

$$\tilde{\mathbf{h}} = H^T \omega \tag{29}$$

we can efficiently compute the elements of $K \times m$ matrix A by accessing only the rows of V whose indexes are in \mathcal{K} as follows:

$$A_{kj} = (\tilde{H}_j)^T V_{\mathcal{K}_{kj}, \bullet} + \tilde{\mathbf{h}}_j \tag{30}$$

4.5.2 Gradient ∇_H

Let us now adapt the computation of the gradient with respect to H , starting from previous Eq. 21 i.e. $\nabla_H = 2QH \text{diag}(\nabla_q) + \hat{Z}$ with $\hat{Z} = \bar{w}\nabla_s^T + W^T\dot{Y}$.

Supposing we have kept an up-to-date Q and \bar{w} (we shall see how to update them effectively in the section 4.6), we are left with only adapting the computation of the $W^T\dot{Y}$ term to use the factored representation of W :

$$\begin{aligned}
\hat{Z} &= \bar{w}\nabla_s^T + W^T\dot{Y} \\
&= \bar{w}\nabla_s^T + (VU + \mathbf{1}_D\omega^T)^T \dot{Y} \\
&= \bar{w}\nabla_s^T + U^T V^T \dot{Y} + \omega \mathbf{1}_D^T \dot{Y} \\
&= \bar{w}\nabla_s^T + U^T (V^T \dot{Y}) + \omega (\dot{Y}^T \mathbf{1}_D)^T \\
&= \bar{w}\nabla_s^T + U^T (V^T \dot{Y}) + \omega \bar{\mathbf{y}}^T
\end{aligned} \tag{31}$$

provided we defined

$$\bar{\mathbf{y}} = \dot{Y}^T \mathbf{1}_D = \text{rowsum}(\dot{Y}) = \text{rowsum}(\nabla_A) \tag{32}$$

We see that computing $d \times m$ matrix \hat{Z} in this manner can be achieved efficiently using our factored representation V, U and ω . Note that computing $V^T \dot{Y}$ is a multiplication by sparse matrix \dot{Y} which will have a computational complexity of $O(Kdm)$, and yield a $d \times m$ matrix. The computation of \hat{Z} in this manner thus has $aO(dm + d^2m + Kdm + dm)$ complexity.

We can then proceed to computing ∇_H as in Eq. 21:

$$\nabla_H = 2 \underbrace{QH}_{\hat{H}} \text{diag}(\nabla_q) + \hat{Z} \tag{33}$$

4.6 Bookkeeping operations: keeping up-to-date \bar{w} and Q

We have shown in section 4.4 that our updates to V, U, ω (Eq. 27,25,26) achieve the same update on (an implicit) W as Eq. 23, i.e. $W_{new} = W - \eta \left(2WH \text{diag}(\nabla_q) + \mathbf{1}_D \nabla_s^T + \dot{Y} \right) H^T$. The efficient computation of loss L and gradient ∇_H seen in Section 4.5 relies on having an up-to-date $Q = W^T W$ and $\bar{w} = \text{rowsum}(W) = (\text{sum}(W_1), \dots, \text{sum}(W_d)) = W^T \mathbf{1}_D$. In this section, we derive efficient updates to \bar{w} and Q that reflect the update to W .

4.6.1 Update of \bar{w}

$$\begin{aligned}
\bar{w}_{new} &= W_{new}^T \mathbf{1}_D \\
&= \left(W - \eta \left(2WH \text{diag}(\nabla_q) + \mathbf{1}_D \nabla_s^T + \dot{Y} \right) H^T \right)^T \mathbf{1}_D \\
&= W^T \mathbf{1}_D - \eta H \left(2WH \text{diag}(\nabla_q) + \mathbf{1}_D \nabla_s^T + \dot{Y} \right)^T \mathbf{1}_D
\end{aligned}$$

$$\begin{aligned}
&= \bar{w} - \eta H \left(2\text{diag}(\nabla_q) H^T W^T + \nabla_s \mathbf{1}_D^T + \dot{Y}^T \right) \mathbf{1}_D \\
&= \bar{w} - 2\eta H \text{diag}(\nabla_q) H^T W^T \mathbf{1}_D - \eta H \nabla_s \mathbf{1}_D^T \mathbf{1}_D - \eta H \underbrace{\dot{Y}^T \mathbf{1}_D}_{\bar{y}} \\
&= \bar{w} - 2\eta H \text{diag}(\nabla_q) H^T \bar{w} - \eta D H \nabla_s - \eta H \bar{y} \\
&= \bar{w} - \eta H \left(2\text{diag}(\nabla_q) H^T \bar{w} - \eta D \nabla_s - \eta \bar{y} \right) \tag{34}
\end{aligned}$$

4.6.2 Update of Q

$$\begin{aligned}
Q_{new} &= W_{new}^T W_{new} \\
&= (W - \eta \nabla_O H^T)^T (W - \eta \nabla_O H^T) \\
&= W^T W - W^T (\eta \nabla_O H^T) - (\eta \nabla_O H^T)^T W + \eta^2 (\nabla_O H^T)^T \nabla_O H^T \\
&= \underbrace{W^T W}_Q - \eta \underbrace{W^T \nabla_O}_{\nabla_H} H^T - \eta \underbrace{(W^T \nabla_O)^T}_{\nabla_H} H^T + \eta^2 H \nabla_O^T \nabla_O H^T \\
Q_{new} &= Q - \eta (\nabla_H H^T) - \eta (\nabla_H H^T)^T + \eta^2 H \underbrace{(\nabla_O^T \nabla_O)}_M H^T \tag{35}
\end{aligned}$$

where we used the fact that $W^T W = Q$ and $\nabla_H = W^T \nabla_O$. Note that while computing $\nabla_O H^T$ would be a prohibitive $O(mDd)$ computation (in addition to requiring to explicitly compute ∇_O in the first place), computing $\nabla_H H^T$ is a comparatively cheap $O(md^2)$ operation.

It remains to derive a way to efficiently compute $m \times m$ matrix $M = \nabla_O^T \nabla_O$ without explicitly computing O nor resorting to explicit W . Substituting ∇_O by its expression from Eq. 19 i.e. $\nabla_O = 2O \text{diag}(\nabla_q) + \mathbf{1}_D \nabla_s^T + \dot{Y}$ yields

$$\begin{aligned}
M &= \nabla_O^T \nabla_O \\
M &= \left(2O \text{diag}(\nabla_q) + \mathbf{1}_D \nabla_s^T + \dot{Y} \right)^T \left(2O \text{diag}(\nabla_q) + \mathbf{1}_D \nabla_s^T + \dot{Y} \right) \\
M &= \left((2O \text{diag}(\nabla_q))^T + (\mathbf{1}_D \nabla_s^T + \dot{Y})^T \right) \left((2O \text{diag}(\nabla_q)) + (\mathbf{1}_D \nabla_s^T + \dot{Y}) \right) \\
M &= (2O \text{diag}(\nabla_q))^T (2O \text{diag}(\nabla_q)) + (\mathbf{1}_D \nabla_s^T + \dot{Y})^T (\mathbf{1}_D \nabla_s^T + \dot{Y}) \\
&\quad + (2O \text{diag}(\nabla_q))^T (\mathbf{1}_D \nabla_s^T + \dot{Y}) + (\mathbf{1}_D \nabla_s^T + \dot{Y})^T (2O \text{diag}(\nabla_q)) \\
M &= (4\text{diag}(\nabla_q) O^T O \text{diag}(\nabla_q)) + \left(\nabla_s \mathbf{1}_D^T \mathbf{1}_D \nabla_s^T + \dot{Y}^T \dot{Y} + \nabla_s \mathbf{1}_D^T \dot{Y} + \dot{Y}^T \mathbf{1}_D \nabla_s^T \right) \\
&\quad + (2O \text{diag}(\nabla_q))^T (\mathbf{1}_D \nabla_s^T + \dot{Y}) + (\mathbf{1}_D \nabla_s^T + \dot{Y})^T (2O \text{diag}(\nabla_q)) \\
M &= 4\text{diag}(\nabla_q) O^T O \text{diag}(\nabla_q) + \left(D \nabla_s \nabla_s^T + \dot{Y}^T \dot{Y} + \nabla_s \bar{y}^T + \bar{y} \nabla_s^T \right) \\
&\quad + (2O \text{diag}(\nabla_q))^T (\nabla_s \mathbf{1}_D^T + \dot{Y}^T) + (\nabla_s \mathbf{1}_D^T + \dot{Y}^T) (2O \text{diag}(\nabla_q))
\end{aligned}$$

$$\begin{aligned}
M &= 4\text{diag}(\nabla_q)O^TO\text{diag}(\nabla_q) + \left(D\nabla_s\nabla_s^T + \mathring{Y}^T\mathring{Y} + \nabla_s\bar{\mathbf{y}}^T + \bar{\mathbf{y}}\nabla_s^T\right) \\
&\quad + \left(\left(\nabla_s\mathbf{1}_D^T + \mathring{Y}^T\right)(2O\text{diag}(\nabla_q))\right)^T + \left(\left(\nabla_s\mathbf{1}_D^T + \mathring{Y}^T\right)(2O\text{diag}(\nabla_q))\right) \\
M &= 4\text{diag}(\nabla_q)O^TO\text{diag}(\nabla_q) + \left(D\nabla_s\nabla_s^T + \mathring{Y}^T\mathring{Y} + \nabla_s\bar{\mathbf{y}}^T + \bar{\mathbf{y}}\nabla_s^T\right) \\
&\quad + \left(2\nabla_s\mathbf{1}_D^TO\text{diag}(\nabla_q) + 2\mathring{Y}^TO\text{diag}(\nabla_q)\right)^T + \left(2\nabla_s\mathbf{1}_D^TO\text{diag}(\nabla_q) + 2\mathring{Y}^TO\text{diag}(\nabla_q)\right)
\end{aligned}$$

Since $O = WH$ we have $O^TO = H^TW^TWH = H^TQH$ and $\mathbf{1}_D^TO = \mathbf{1}_D^TWH = \bar{w}^TH$. Substituting these in the above expression of M we obtain

$$\begin{aligned}
M &= 4\text{diag}(\nabla_q)\overbrace{O^TO}^{H^TQH}\text{diag}(\nabla_q) + (D\nabla_s\nabla_s^T + \overbrace{\mathring{Y}^T\mathring{Y}}^{\hat{M}} + \nabla_s\bar{\mathbf{y}}^T + \bar{\mathbf{y}}\nabla_s^T) \\
&\quad + (2\nabla_s\overbrace{\mathbf{1}_D^TO}^{\bar{w}^TH}\text{diag}(\nabla_q) + 2\mathring{Y}^T\overbrace{O}^{WH}\text{diag}(\nabla_q))^T + (2\nabla_s\overbrace{\mathbf{1}_D^TO}^{\bar{w}^TH}\text{diag}(\nabla_q) + 2\mathring{Y}^T\overbrace{O}^{WH}\text{diag}(\nabla_q)) \\
M &= 4\text{diag}(\nabla_q)H^TQH\text{diag}(\nabla_q) + \left(D\nabla_s\nabla_s^T + \mathring{Y}^T\mathring{Y} + \nabla_s\bar{\mathbf{y}}^T + \bar{\mathbf{y}}\nabla_s^T\right) \\
&\quad + 2\left(\nabla_s\bar{w}^TH\text{diag}(\nabla_q) + \mathring{Y}^TWH\text{diag}(\nabla_q)\right)^T + 2\left(\nabla_s\bar{w}^TH\text{diag}(\nabla_q) + \mathring{Y}^TWH\text{diag}(\nabla_q)\right) \\
M &= 4\text{diag}(\nabla_q)H^TQH\text{diag}(\nabla_q) + \left(D\nabla_s\nabla_s^T + \mathring{Y}^T\mathring{Y} + \nabla_s\bar{\mathbf{y}}^T + \bar{\mathbf{y}}\nabla_s^T\right) \\
&\quad + 2\left(\left(\nabla_s\bar{w}^T + \mathring{Y}^TW\right)H\text{diag}(\nabla_q)\right)^T + 2\left(\left(\nabla_s\bar{w}^T + \mathring{Y}^TW\right)H\text{diag}(\nabla_q)\right) \\
M &= 4\text{diag}(\nabla_q)H^TQH\text{diag}(\nabla_q) + \left(D\nabla_s\nabla_s^T + \mathring{Y}^T\mathring{Y} + \nabla_s\bar{\mathbf{y}}^T + \bar{\mathbf{y}}\nabla_s^T\right) \\
&\quad + 2(\text{diag}(\nabla_q)H^T\overbrace{(\bar{w}\nabla_s^T + W^T\mathring{Y})}^{\hat{Z}} + 2(\text{diag}(\nabla_q)H^T\overbrace{(\bar{w}\nabla_s^T + W^T\mathring{Y})}^{\hat{Z}}))^T.
\end{aligned}$$

Reusing previously defined $\hat{Z} = \bar{w}\nabla_s^T + W^T\mathring{Y}$ that were already part of the computation of ∇_H (see Eq. 33 in section 4.5.2), we can thus compute M efficiently as

$$\begin{aligned}
M &= 4\text{diag}(\nabla_q)\overbrace{H^TQH}^{\hat{M}}\text{diag}(\nabla_q) + \left(D\nabla_s\nabla_s^T + \mathring{Y}^T\mathring{Y} + \nabla_s\bar{\mathbf{y}}^T + \bar{\mathbf{y}}\nabla_s^T\right) \\
&\quad + 2\left(\text{diag}(\nabla_q)H^T\hat{Z}\right) + 2\left(\text{diag}(\nabla_q)H^T\hat{Z}\right)^T
\end{aligned} \tag{36}$$

Note that computing M requires computing $\mathring{Y}^T\mathring{Y}$, a $m \times m$ matrix, each element of which is the dot product between two K - *sparse* columns of sparse matrix \mathring{Y} so that it can be computed in $O(m^2K)$.

Having M we can then update Q using Eq. 35.

4.7 Bookkeeping operations: tracking U^{-T}

We can update U^{-T} to reflect our rank- m update of U in step a), using the Woodbury identity.

4.8 Putting it all together

In this section, we put together all the operations that we have derived to write the minibatch version of the update algorithm for general spherical losses.

The parameters of the output layer that we will learn are V, U, ω and implicitly represent W as $W = VU + \mathbf{1}_D \omega^T$.

The algorithm will work for any spherical loss function ℓ in canonical form that computes $\ell(q, s, \mathcal{K}, \mathbf{a}, \mathbf{t})$ and for which we can compute gradients with respect to its parameters.

Initialization

- we can initialize $D \times d$ matrix V randomly as we would have initialized W so that we initially have $V = W$.
Alternatively we can initialize V to 0 (there won't be symmetry breaking issues with having W initially be 0 provided the other layers are initialized randomly, since varying inputs and targets will naturally break symmetry for the output layer)
- we initialize U to the identity: $U \leftarrow \mathbf{I}_d$
- and ω to zero $\omega \leftarrow \mathbf{0}_d$ so that, trivially, we initially have $VU + \mathbf{1}_D \omega^T = W$.
- initialize $U^{-T} \leftarrow \mathbf{I}_d$
- initialize $Q \leftarrow W^T W = V^T V$ (or more cheaply initialize $Q \leftarrow 0$ if we have initialized V to 0).
- initialize $\bar{w} = W^T \mathbf{1}_D = \text{rowsum}(W) = \text{rowsum}(V)$ (or more cheaply $\bar{w} \leftarrow 0$ if we have initialized V to 0).

Minibatch update algorithm for arbitrary spherical loss

Inputs (besides above parameters V, U, ω and bookkeeping variables Q, U^{-T}, \bar{w}):

- H : a $d \times m$ matrix whose m columns contain the last hidden layer representation vectors for m example (with an appended constant 1 element to account for an output bias).
- Y : a $D \times m$ sparse target matrix that uses sparse representation (\mathcal{K}, T) so that $Y_{\mathcal{K}_{kj}, j} = T_{kj}$ for $k \in \{1, \dots, K\}$ and $j \in \{1, \dots, m\}$. Each of the m columns of Y is the K -sparse target vector associated to one example of the minibatch.
- $\eta \in \mathbb{R}^+$ learning rate for the update

Updates:

- parameters and bookkeeping matrices $U, V, \omega, Q, U^{-T}, \bar{w}$

Returns:

- $L \in \mathbb{R}$ the sum of squared error losses for the m examples of the minibatch
- ∇_H a $d \times m$ matrix whose m columns contain the gradient of the loss with respect to H , to further backpropagate upstream.

The detailed algorithm is given as Algorithm 4

Counting the total number of basic operations of the update algorithm yields roughly $8md^2 + m^3 + 7m^2d + 2mKd + 3d^2 \approx 17md^2$ operations.

Comparing this $17md^2$ to the $3Dm$ of the naive update, the expected theoretical speedup is approximately $\frac{3D}{18d} = \frac{1}{6} \frac{D}{d}$

For $d = 512$ and $D = 793471$ this yields a theoretical speedup of 258

Note that in the special cases where the specific loss function ℓ does not depend on the sum of outputs s (as is the case e.g. of the squared error) then we don't need to compute s , and can use $a\omega$ that is always 0 so there's a lot we don't need to compute and update.

5 Controlling numerical stability

The update of U may over time lead to U becoming ill-conditioned. Simultaneously, as we update U and U^{-T} (using Sherman-Morrison or Woodbury) our updated U^{-T} may over time start to diverge from the true U^{-T} due to numerical precision. It is thus important to prevent both of these from happening, i.e. make sure U stays well conditioned, to ensure the numerical stability of the algorithm. We present here progressively refined strategies for achieving this.

5.1 Restoring the system in a pristine stable state

One simple way to ensure numerical stability is to once in a while restore the system in its pristine state where $V = W$ and $U = \mathbf{I}_d = U^{-T}$. This is easily achieved as follows:

$$\begin{aligned} V &\leftarrow VU \\ U &\leftarrow \mathbf{I}_d \\ U^{-T} &\leftarrow \mathbf{I}_d. \end{aligned}$$

This operation doesn't affect the product VU , so the implicit matrix W remains unchanged, nor does it affect $Q = W^T W$. And it does restore U to a perfectly well conditioned identity matrix. But computing VU is an extremely costly $O(Dd^2)$ operation, so if possible we want to avoid it (except maybe once at the very end of training, if we want to compute the actual W). In the next paragraphs we develop a more efficient strategy.

5.2 Stabilizing only problematic singular values

U becoming ill-conditioned is due to its singular values over time becoming too large and/or too small. Let us define $\sigma_1, \dots, \sigma_d$ as the singular values of U ordered in decreasing order. The conditioning number of U is defined as $\frac{\sigma_1}{\sigma_d}$ and it can become overly large when σ_1 becomes too large and/or when σ_d becomes too small. Restoring

Algorithm 4 Minibatch version of the update algorithm for general spherical loss

FUNCTION spherical_minibatch_fbprop_update:

	hidden layer minibatch	sparse target	learning rate	layer parameters	bookkeeping variables
Inputs:	\widehat{H}	$\widehat{\mathcal{K}}, \widehat{T}$	$\widehat{\eta}$	$\widehat{V}, \widehat{U}, \widehat{\omega}$	$\widehat{Q}, \widehat{\bar{w}}, \widehat{U}^{-T}$
Updates:	$V, U, \omega, Q, \bar{w}, U^{-T}$				
Returns:	loss L , gradient ∇_H to backpropagate further upstream				
Operations	main	result	# ops		
	text Eq.	dims			
$\hat{H} = QH$	Eq. 14	$d \times m$	md^2		
$\hat{M} = H^T \hat{H}$	Eq. 14	$m \times m$			
$\mathbf{q} = \text{diag}(\hat{M})$	Eq. 14	m	m		
$\mathbf{s} = H^T \bar{w}$	Eq. 15	m	md		
$\tilde{H} = UH$	Eq. 28	$d \times m$	md^2		
$\tilde{\mathbf{h}} = H^T \omega$	Eq. 29	m	md		
Matrix $A: A_{kj} = (\tilde{H}_j)^T V_{\mathcal{K}_{kj}} \bullet + \tilde{\mathbf{h}}_j$	Eq. 30	$K \times m$	mKd		
$\tilde{L} = [\ell(\mathbf{q}_j, \mathbf{s}_j, \mathcal{K}_j, A_j, T_j)]_{j=1\dots m}$	Eq. 16	m	typically $O(Km)$		
$L = \text{sum}(\tilde{L})$		1	m		
$\nabla_q = \left[\frac{\partial \ell}{\partial q}(\mathbf{q}_j, \mathbf{s}_j, \mathcal{K}_j, A_j, T_j) \right]_{j=1\dots m}$		m			
$\nabla_s = \left[\frac{\partial \ell}{\partial s}(\mathbf{q}_j, \mathbf{s}_j, \mathcal{K}_j, A_j, T_j) \right]_{j=1\dots m}$		m			
$\nabla_A = \left[\frac{\partial \ell}{\partial \mathbf{a}_k}(\mathbf{q}_j, \mathbf{s}_j, \mathcal{K}_j, A_j, T_j) \right]_{k=1\dots K, j=1\dots m}$		$K \times m$			
$\hat{Y} = \text{sparsemat}_{D,m}(\mathcal{K}, \nabla_A)$		$D \times m$ (K -sparse)			
$\bar{\mathbf{y}} = \hat{Y}^T \mathbf{1}_D = \text{rowsum}(\nabla_A)$	Eq. 32	m	Km		
$\hat{Z} = \bar{w} \nabla_s^T + U^T (V^T \hat{Y}) + \omega \bar{\mathbf{y}}^T$	Eq. 31	$d \times m$	md		
$\nabla_H = 2\hat{H} \text{diag}(\nabla_q) + \hat{Z}$	Eq. 33	$d \times m$	md		
$U \leftarrow U - 2\eta \underbrace{(UH)}_{\tilde{H}} \text{diag}(\nabla_q) H^T$	Eq. 25	$d \times d$	md^2		
$U^{-T} \leftarrow \dots$ use Woodbury Identity to update it.		$d \times d$	$2m^2d + m^3 + 2md^2$		
$\omega \leftarrow \omega - \eta H(2\text{diag}(\nabla_q) \underbrace{H^T}_{\tilde{\mathbf{h}}} \omega + \nabla_s)$	Eq. 26	d	$2md + 3d$		
$V \leftarrow V - \eta \hat{Y} (U_{new}^{-T} H)^T$	Eq. 27	$D \times d$	$md^2 + mKd$		
$\bar{w} \leftarrow \bar{w} - \eta H(2\text{diag}(\nabla_q) H^T \bar{w} + D \nabla_s + \bar{\mathbf{y}})$	Eq. 34	d	$2md + 4d$		
$M = 4\text{diag}(\nabla_q) \hat{M} \text{diag}(\nabla_q) + D \nabla_s \nabla_s^T + \hat{Y}^T \hat{Y} + \nabla_s \bar{\mathbf{y}}^T + \bar{\mathbf{y}} \nabla_s^T + 2 \left(\text{diag}(\nabla_q) H^T \hat{Z} \right) + 2 \left(\text{diag}(\nabla_q) H^T \hat{Z} \right)^T$	Eq. 36	$m \times m$	$2m^2d + (5 + K)m^2 + d^2$		
$Q \leftarrow Q - \eta \nabla_H H^T - \eta H \nabla_H^T + \eta^2 (HM) H^T$	Eq. 35	$d \times d$	$md^2 + 2m^2d + 2d^2$		

RETURN L, ∇_H

the system in its pristine state, as shown in the previous paragraph, in effect brings back *all* singular values of U back to 1 (since it brings back U to being the identity). It is instead possible, and computationally far less costly, to correct when needed only for the singular values of U that fall outside a safe range. Most often we will only need to occasionally correct for one singular value (usually the smallest, and only when it becomes too small). Once we have determined the offending singular value and its corresponding singular vectors, correcting for that singular value, i.e. effectively bringing it back to 1, will be a $O(Dd)$ operation. The point is to apply corrective steps only on the problematic singular values and only when needed, rather than blindly, needlessly and inefficiently correcting for all of them through the basic $O(Dd^2)$ full restoration explained in the previous paragraph.

Here is the detailed algorithm that achieves this:

Algorithm 5 Numerical stabilization procedure for problematic singular values

- The chosen safe range for singular values is $[\sigma_{\text{low}}, \sigma_{\text{high}}]$ (ex: $[0.001, 100]$)
- The procedures given below act on output layer parameters U , U^{-T} and V .
- For concision, we do not enlist these parameters explicitly in their parameter list.
- Procedure SINGULAR-STABILIZE gets called after every n_{check} gradient updates (ex: $n_{\text{check}} = 100$).

procedure SINGULAR-STABILIZE()

$\bar{U}, \sigma, \bar{V} = \text{SVD}(U)$ ▷ Computes singular value decomposition of U as
 $U = \bar{U} \text{diag}(\sigma) \bar{V}^T$
for all $k \in \{1, \dots, d\}$ **do**
 if $\sigma_k < \sigma_{\text{low}}$ **OR** $\sigma_k > \sigma_{\text{high}}$ **then**
 FIX-SINGULAR-VALUE($\sigma_k, \bar{U}_k, 1$)
 end if
end for
end procedure

The following procedure will change singular value σ of U associated to singular vector u to become target singular value σ^ (typically 1). It doesn't change U 's singular vectors, only that one singular value. It also changes V symmetrically (with a rank-one update) in such a way that $W = VU$ remains unchanged.*

procedure FIX-SINGULAR-VALUE(σ, u, σ^*)

$\alpha = \frac{\sigma^* - \sigma}{\sigma}$
 $\beta = -\frac{\alpha}{1 + \alpha}$
 $U \leftarrow U + \alpha u (U^T u)^T$
 $V \leftarrow V + \beta (V u) u^T$
 $U^{-T} \leftarrow U^{-T} + \beta u (U^{-1} u)^T$ ▷

Where U^{-1} is obtained as the transpose of U^{-T} . But we may instead of this prefer to recompute U^{-T} from scratch by inverting U to ensure it doesn't stray too much due to numerical imprecisions.

end procedure

Proof that $W = VU$ is left unchanged by FIX-SINGULAR-VALUE

$$\begin{aligned}
V_{new}U_{new} &= (V + \beta(Vu)u^T)(U + \alpha u(U^T u)^T) \\
&= V(\mathbf{I}_d + \beta uu^T)(U + \alpha uu^T U) \\
&= V(\mathbf{I}_d + \beta uu^T)(\mathbf{I}_d + \alpha uu^T)U \\
&= V(\mathbf{I}_d^2 + \beta uu^T + \alpha uu^T + \beta \alpha uu^T uu^T)U \\
&= V(\mathbf{I}_d^2 + (\alpha + \beta)uu^T + \beta \alpha u(u^T u)u^T)U \\
&= V(\mathbf{I}_d + (\alpha + \beta)uu^T + \beta \alpha uu^T)U \\
&= V(\mathbf{I}_d + (\alpha - \frac{\alpha}{1 + \alpha} + \alpha \frac{-\alpha}{1 + \alpha})uu^T)U \\
&= V(\mathbf{I}_d + (\alpha - \frac{\alpha}{1 + \alpha} - \frac{\alpha^2}{1 + \alpha})uu^T)U \\
&= V(\mathbf{I}_d + (\alpha - \frac{\alpha + \alpha^2}{1 + \alpha})uu^T)U \\
&= V(\mathbf{I}_d + (\alpha - \frac{\alpha(1 + \alpha)}{1 + \alpha})uu^T)U \\
&= V(\mathbf{I}_d + (\alpha - \alpha)uu^T)U \\
&= V\mathbf{I}_d U \\
&= VU
\end{aligned}$$

5.3 Avoiding the cost of a full singular-value decomposition

Computing the SVD of $d \times d$ matrix U as required above, costs roughly $25d^3$ elementary operations (use the so-called R-SVD algorithm). But since the offending singular values will typically be only the smallest or the largest, it is wasteful to compute all d singular values every time. A possibly cheaper alternative is to use the power iteration method with U to find its largest singular value and associated singular vector, and similarly with U^{-1} to obtain the smallest singular value of U (which corresponds to the inverse of the largest singular value of U^{-1}). Each iteration of the power iteration method requires only $O(d^2)$ operations, and a few iterations may suffice. In our experiments we fixed it to 100 power iterations. Also it is probably not critical if the power iteration method is not run fully to convergence, as correcting along an approximate offending singular vector direction may be sufficient for the purpose of ensuring numerical stability.

With this refinement, we loop over finding the smallest singular value with the power iteration method, correcting for it to be 1 by calling FIX-SINGULAR-VALUE if it is too small, and we repeat this until we find the now smallest singular value to be inside the acceptable range. Similarly for the largest singular values.

Note that while in principle we may not need to ever invert U from scratch (as we provided update formulas of U^{-T} with every change we make to U), it nevertheless proved to be necessary to do so regularly to ensure U^{-T} doesn't stray too much from the correct value due to numerical imprecisions. Inverting U using Gaussian-elimination costs roughly d^3 operations, so it is very reasonable and won't affect the

Table 1: Speedups with respect to the baseline naive model on CPU, for a minibatch of 128 and the whole vocabulary of $D = 793471$ words. This is a two hidden layer model with 300 neurons on all its layers (so $d = 300$).

Model	output layer only speedup	whole model speedup
cpu unfactorised (naive)	1	1
gpu unfactorised (naive)	6.8	4.7
gpu hierarchical softmax	125.2	178.1
cpu factorised	763.3	501
gpu factorised	3257.3	1852.3

computational complexity if we do it no more often than every d training examples (which will typically correspond to less than 10 minibatches of size 128). In practice, we recompute U^{-T} from scratch every time before we run this check for singular value stabilization.

6 Experimental validation

We implemented both a CPU version using *blas* and a parallel GPU (Cuda) version using *cublas* of the proposed algorithm⁴. We evaluated the GPU and CPU implementations by training word embeddings with simple neural language models, in which a probability map of the next word given its preceding n -gram is learned by a neural network. We used a Nvidia Titan Black GPU and a i7-4820K @ 3.70GHz CPU and ran experiments on the one billion word dataset[?], which is composed of 0.8 billions words belonging to a vocabulary of 0.8 millions words. We evaluated the resulting word embeddings with the recently introduced Simlex-999 score [?], which measures the similarity between words. We also compared our approach to unfactorised versions and to a two-layer hierarchical softmax. Figure 2 and 3 (left) illustrate the practical speedup of our approach for the output layer only. Figure 3(right) shows that the LST (Large Sparse Target) models are much faster to train than the softmax models and converge to only slightly lower Simlex-999 scores. Table 1 summarizes the speedups for the different output layers we tried, both on CPU and GPU. We also empirically verified that our proposed factored algorithm learns the exact same model weights (VU) as the corresponding naive unfactored algorithm’s W , as it theoretically should (up to negligible numerical precision differences), and followed the exact same learning curves (as a function of number of iterations, not time!).

7 Conclusion and future work

We introduced a new algorithmic approach to efficiently compute the *exact* gradient updates for training deep networks with very large sparse targets. Remarkably the complexity of the algorithm is independent of the target size, which allows tackling

⁴Open source code will be released upon official publication of this research.

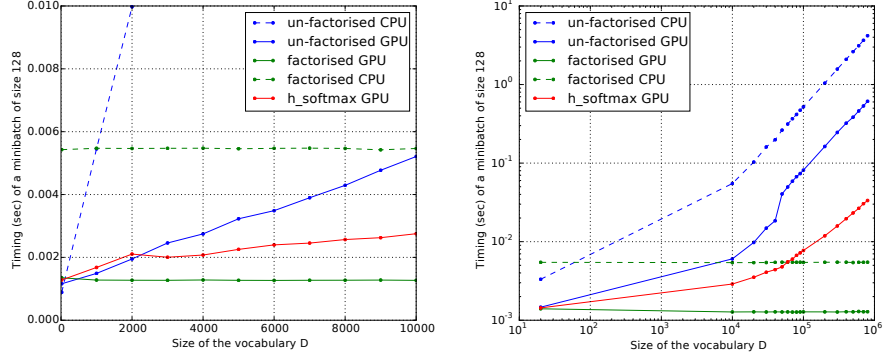


Figure 2: Timing of different algorithms. Time taken by forward and backward propagations in the output layer, including weight update, on a minibatch of size 128 for different sizes of vocabulary D on both CPU and GPU. The input size d is fixed to 300. The Timing of a 2 layer hierarchical softmax efficient GPU implementation (h_softmax) is also provided for comparison. Right plot is in log-log scale. As expected, the timings of factorized versions are independent of the size of the vocabulary.

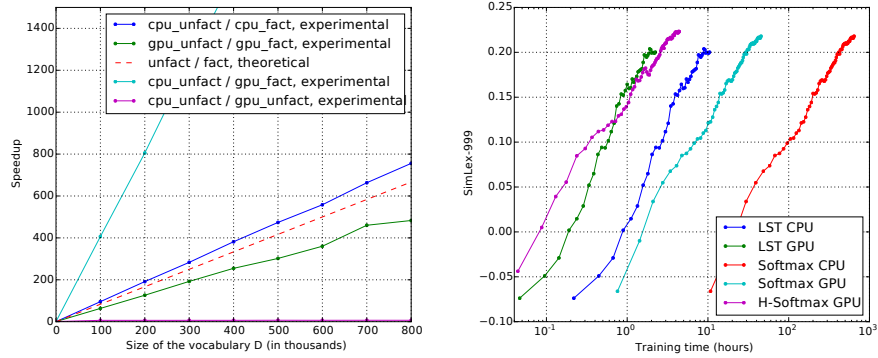


Figure 3: **Left:** Practical and theoretical speedups for different sizes of vocabulary D and fixed input size $d=300$. The practical unifact / fact speedup is similar to the theoretical one. **Right:** Evolution of the Simlex-999 score obtained with different models as a function of training time (CPU softmax times were extrapolated from fewer iterations). Softmax models are zero hidden-layer models, while our large sparse target (LST) models have two hidden layers. These were the best architectures retained in both cases (surprisingly the softmax models with hidden layers performed no better on this task). The extra non-linear layers in LST may help compensate for the lack of a softmax. LST models converge to slightly lower scores at similar speed as the hierarchical softmax model but significantly faster than softmax models.

very large problems. Our CPU and GPU implementation yield similar speedups to the theoretical one and can thus be used in practical applications, which could be explored in further work. In particular, neural language models seem good candidates. But it remains unclear how using a loss function other than *log-softmax* may affect the quality of the resulting word embeddings and further research should be carried out in this direction. While restricted, the spherical family of loss functions, offers opportunities to explore alternatives to the ubiquitous softmax, that thanks to the algorithm presented here, could scale computationally to extremely large output spaces.

Acknowledgements

We would like to thank the developers of Theano [14, 15] and Blocks [16].

This research is supported by NSERC and Ubisoft.

References

- [1] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A neural probabilistic language model. In *NIPS'00*, pages 932–938. MIT Press, 2001.
- [2] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12: 2493–2537, 2011.
- [3] Y. Dauphin, X. Glorot, and Y. Bengio. Large-scale learning of embeddings with reconstruction sampling. In *Proceedings of the 28th International Conference on Machine learning, ICML '11*, 2011.
- [4] Sébastien Jean, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. On using very large target vocabulary for neural machine translation. In *ACL-IJCNLP'2015*, 2015. arXiv:1412.2007.
- [5] M. Gutmann and A. Hyvarinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of The Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS'10)*, 2010.
- [6] Andriy Mnih and Koray Kavukcuoglu. Learning word embeddings efficiently with noise-contrastive estimation. In C.J.C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2265–2273. Curran Associates, Inc., 2013.
- [7] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS'2013*, pages 3111–3119. 2013.
- [8] Anshumali Shrivastava and Ping Li. Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS). In Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2321–2329. Curran Associates, Inc., 2014.
- [9] Sudheendra Vijayanarasimhan, Jonathon Shlens, Rajat Monga, and Jay Yagnik. Deep networks with large output spaces. arxiv:1412.7479, 2014.

- [10] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In Robert G. Cowell and Zoubin Ghahramani, editors, *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics*, pages 246–252. Society for Artificial Intelligence and Statistics, 2005.
- [11] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.
- [12] Yann LeCun. Une procédure d’apprentissage pour Réseau à seuil assymétrique. In *Cognitive 85: A la Frontière de l’Intelligence Artificielle, des Sciences de la Connaissance et des Neurosciences*, pages 599–604, Paris 1985, 1985. CESTA, Paris.
- [13] Yann LeCun. Learning processes in an asymmetric threshold network. In E. Bienenstock, F. Fogelman-Soulié, and G. Weisbuch, editors, *Disordered Systems and Biological Organization*, pages 233–240. Springer-Verlag, Berlin, Les Houches 1985, 1986.
- [14] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010. Oral Presentation.
- [15] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.
- [16] B. van Merriënboer, D. Bahdanau, V. Dumoulin, D. Serdyuk, D. Warde-Farley, J. Chorowski, and Y. Bengio. Blocks and Fuel: Frameworks for deep learning. *ArXiv e-prints*, jun 2015.